A Star* Python Implementation

CIS 479

Project 1 Report

Team: Paul Murariu &
Dominic Baughman

Date: 9/24/2023

```
## ## ## 00 ## ## ## ## ## ## ## ##
## 07 02 01 03 ## 48 49 50 51 52 ##
## 08 ## ## 04 ## 46 ## ## ## 53 ##
## 09 ## 06 05 ## 43 40 38 ## 54 55
## 10 ## ## ## ## ## ## 36 ## ## ##
## 11 12 ## 19 20 21 ## 35 37 39 ##
## ## 13 ## 18 ## ## ## 33 ## 41 ##
## 15 14 16 17 ## 30 31 32 ## 44 ##
## 22 ## ## ## ## 29 ## 34 ## ## ##
## 23 24 25 26 27 28 ## 42 45 47 ##
## ## ## ## ## ## ## ## ## ## ## ##
```

7:20 PM
9/24/2023

Programmed by pmurariu and baughboy

```python
# Maze and Path - done by pmurariu and baughboy
from queue import PriorityQueue
# 1 = wall
# 0 = path
maze = [
    [1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1],
    [1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1],
    [1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0],
    [1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1],
    [1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1],
    [1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
    [1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
]
path = [
    ["##", "##", "##", "[]" ,"##", "##", "##", "##", "##", "##", "##", "##"],
    ["##", "[]", "[]", "[]", "[]", "##", "[]", "[]", "[]", "[]", "[]", "##"],
    ["##", "[]", "##", "##", "[]", "##", "[]", "##", "##", "##", "[]", "##"],
    ["##", "[]", "##", "[]", "[]", "##", "[]", "[]", "[]", "##", "[]", "[]"],
    ["##", "[]", "##", "##", "##", "##", "##", "##", "[]", "##", "##", "##"],
    ["##", "[]", "[]", "##", "[]", "[]", "[]", "##", "[]", "[]", "[]", "##"],
    ["##", "##", "[]", "##", "[]", "##", "##", "##", "[]", "##", "[]", "##"],
    ["##", "[]", "[]", "[]", "[]", "##", "[]", "[]", "[]", "##", "[]", "##"],
    ["##", "[]", "##", "##", "##", "##", "[]", "##", "[]", "##", "##", "##"],
    ["##", "[]", "[]", "[]", "[]", "[]", "[]", "##", "[]", "[]", "[]", "##"],
    ["##", "##", "##", "##", "##", "##", "##", "##", "##", "##", "##", "##"]
]
```

6:03 PM
9/24/2023

```
# Priority queue for the frontier set - done by pmurariu and baughboy
# Write a PQueue that when you put a step in it, it sorts it by the f value, if the f value is the same, then sort by the step.label
class StepPQueue:
    queue = []
    def __init__(self, step):
        self.queue.append(step)
    def put(self, step):
        self.queue.append(step)
        self.queue.sort(key=lambda x: (x.f, x.label))
    def get(self):
        return self.queue.pop(0)

# Done by pmurariu and baughboy
# Preset list of directions going west, north, east, south
directions = [(0,-1), (-1, 0), (0,1), (1,0)]
directionCost = [2, 3, 2, 1]
goal_x = 3
goal_y = 11
start_x = 0
start_y = 3
curLabel = "00"
```

6:09 PM
9/24/2023

```
# Done by pmurariu and baughboy
# Define a class named Step
class Step:
    label = ""
    f = 0
    g = 0
    h = 0
    x = 0
    y = 0
    parent = None
    # Done by pmurariu and baughboy
    # Step should have a constructor that takes in a label, g, h, and parent
    def __init__(self, g, h, f, x, y, parent):
        self.label = curLabel
        self.g = g
        self.h = h
        self.f = f
        self.x = x
        self.y = y
        self.parent = parent
# Done by pmurariu and baughboy
def get1Direction(start, end):
    if start[0] - end[0] > 0:
        return 1
    if start[0] - end[0] < 0:
        return 3
    if start[1] - end[1] > 0:
        return 0
    if start[1] - end[1] < 0:
        return 2
    return ""
# start = (0, 3)
# end = (3, 11)
```

6:12 PM
9/24/2023

6:46 PM
9/24/2023

```python
# Done by pmurariu and baughboy
def get2Direction(start, end):
    direction = []
    # Calculate the north/south cost
    if start[0] - end[0] < 0:
        direction.append(3)
    elif start[0] - end[0] > 0:
        direction.append(1)
    else:
        direction.append(3) # if there is no difference just return the south cost direction as a placeholder
    # Calculate the west/east cost
    if start[1] - end[1] < 0:
        direction.append(2)
    elif start[1] - end[1] > 0:
        direction.append(0)
    else:
        direction.append(3) # if there is no difference just return the south cost direction as a placeholder
    return direction

# Done by pmurariu and baughboy
def calculateHeuristic(step):
    # CALCULATE G
    step.g = step.parent.g + directionCost[get1Direction((step.parent.x, step.parent.y), (step.x, step.y))]
    # CALCULATE H
    dCost = get2Direction((step.x, step.y), (goal_x, goal_y))
    step.h = (abs(step.x - goal_x)*directionCost[dCost[0]]) + (abs(step.y - goal_y)*directionCost[dCost[1]])
    # CALCULATE F
    step.f = step.g + step.h
```

```python
# Done by pmurariu and baughboy
def increment():
    global curLabel
    i = int(curLabel)
    i += 1
    temp = str(i)
    if len(temp) == 1:
        label = "0" + temp
        curLabel = label
    else:
        curLabel = temp

# Done by pmurariu and baughboy
def checkValid(maze, step, direction):
    # Check if the step is valid
    if step.x + direction[0] < 0 or step.x + direction[0] > len(maze) - 1:
        return False
    if step.y + direction[1] < 0 or step.y + direction[1] > len(maze[0]) - 1:
        return False
    if maze[step.x + direction[0]][step.y + direction[1]] == 1:
        return False
    return True
```

6:55 PM
9/24/2023

```python
# Done by pmurariu and baughboy
def Astaralgo(maze, step):
    # initialize a PriorityQueue
    frontier = StepPQueue(step)
    visited = {step.label: True}
    while (len(frontier.queue) > 0):
        curPos = frontier.get()
        maze[curPos.x][curPos.y] = 1
        # Add the step to the path
        path[curPos.x][curPos.y] = curPos.label
        visited[curPos.label] = True
        if [curPos.x, curPos.y] == [goal_x, goal_y]:
            return path
        for direction in directions:
            # Check if we can move in the direction
            if checkValid(maze, curPos, direction):
                # Create a new step
                newStep = Step(0, 0, 0, curPos.x + direction[0], curPos.y + direction[1], curPos)
                increment()
                # Calculate Heruistic
                calculateHeuristic(newStep)
                visited[newStep.label] = False
                # Add the step to the frontier
                frontier.put(newStep)
    for cells in path:
        if cell in path == "[]":
            cell = "##"
    return path
```

7:13 PM
9/24/2023

```python
# Done by pmurariu and baughboy
# Create initial step
start_step = Step(0, 0, 0, start_x, start_y, None)
path[start_step.x][start_step.y] = start_step.label
increment()
attempt = Astaralgo(maze, start_step)
# Print the path
for row in attempt:
    # Print the row without the list brackets or commas
    print(" ".join(row))

print()

print("Programmed by pmurariu and baughboy")
```

7:19 PM
9/24/2023

The priority queue for the frontier set is highlighted in yellow where it is implemented using the 'StepPQueue' class. First, the data structure has a constructor that initializes the priority queue with a single step. Next, the 'put' method adds a new step to the 'queue' and sorts the steps according to their ' f ' values. If two steps have the same ' f ' value, then they are ordered by 'label.' The step with the least ' f ' will always be at the top of the list if the queue is sorted each time a new step is added. Finally, the first step in the 'queue', which has the smallest ' f ' value as a result of sorting, is removed and returned by the 'get' method. The 'Astaralgo' function manages the frontier set using this priority queue. The method selects the step from the priority queue with the smallest ' f ' value as the frontier, explores it, and adds any valid neighbors to the frontier. This process continues until the goal is reached or the frontier is empty.

The explored set is using a dictionary data structure and that line of code is the first, yellow highlighted one under def 'Astaralgo(maze, step)' which is the aStar algorithm. The starting node or ('step') is initialized in the 'visited' dictionary when the algorithm starts. The initial node's 'label' is used as the key, and 'True' is set as the value to denote that this node has been visited. As new nodes are discovered by the algorithm (referred to as 'newStep' in the code), and as it does so, it adds them to the 'visited' dictionary with a 'False' value. This means that although they have not been visited or processed yet, these nodes are on the frontier and need to be explored. A node ('curPos') is marked as explored when its corresponding value in the 'visited' dictionary is updated to 'True' after it is removed from the frontier and processed.

My f(n) = g(n) + h(n) is done in the highlighted yellow heuristic function which is implemented in the 'calculateHeuristic' method. To calculate G ('step.g'), the 'g' value for a step is the actual cost from the starting node to the current node ('step'). It

is determined by combining the cost of traveling from the parent node to the current node with the parent node's 'g' value ('step.parent.g'). The 'get1Direction' method is used to retrieve the moving cost from the 'directionCost' array. To calculate H ('step.h'), the 'h' value is the heuristic estimate of the cost from the current node to the goal. To get the directions (north/south and west/east) from the current node to the destination, you use the 'get2Direction' function. The heuristic h is then estimated by multiplying the differences in the x and y coordinates between the current node and the goal by the corresponding direction costs. To calculate F ('step.f'), the ' f ' value is the sum of 'g' and 'h'. It represents the total estimated cost of the cheapest solution through the current node.
'f = g + h.'

Adding leaves into the frontier can be seen highlighted yellow under the aStar algorithm which is the line 'frontier. put(newStep)' specifically. What this line is doing is that it is adding a priority queue as a new step (or state) to the frontier. The frontier contains all the created but unexplored nodes in the context of the aStar algorithm. Each step in the maze represents a potential movement or state.

Picking the smallest f(n) can be seen highlighted yellow under the aStar algorithm which is the line 'curPos = frontier.get().' This line is responsible for dequeuing and then selects the node with the smallest f(n) value from the frontier. The 'get' method of the 'StepPQueue' class retrieves and removes the node with the smallest f(n) value from the priority queue 'frontier'. The successors of this node, 'curPos', are then examined and potentially added to the 'frontier.'